dexyfex.com

Black Holes & Accretion Discs – Gravitational Lensing

Posted on **October 13, 2016** by **dexyfex**

Accretion Discs - Galaxia Procedural Universe T…

Well if you’ve seen the movie **Interstellar (https://en.wikipedia.org/wiki/Interstellar_(film))**, or if you studied any astrophysics at all, you’ll know that black holes look weird because of their huge mass and gravity. The effect could be partly explained just using Newton’s law of gravity, by assuming the mass-energy equivalence as discovered by Einstein (E=mc2, i.e. by assuming the photons are attracted by the gravity), but the correct observed value can only be obtained by the results of Einstein’s **general relativity (https://en.wikipedia.org/wiki/General_relativity)**, accounting for the curvature of space-time around the mass.

Unfortunately though, evaluating the full integral to find the true path of the ray for each pixel seems like it would be computationally prohibitive (i.e. it would need a lot of computing time). So as always in real-time computer graphics, an approximation needs to be made.

A fast, accurate approximation is already made for the gravitational lens for relatively small deflection angles, as a result of finding the limits of the integral. Thanks to Einstein, the deflection angle is approximated as (image from **wikipedia (https://en.wikipedia.org/wiki/Gravitational_lensing_formalism)**) :

$$\hat{\alpha} = \frac{4GM}{c^2 b}$$

where G is the gravitational constant, M is the mass of the object (black hole in this case), c is the speed of light and b is the **impact parameter (https://en.wikipedia.org/wiki/Impact_parameter)**, or the perpendicular distance to the center of the black hole from the viewing ray. Interestingly, this equation was first shown to be correct by Arthur Eddington and Frank Dyson in 1919 during a solar eclipse by observing stars appearing very near the Sun, and measuring how much their light had been deflected by the Sun's mass, thus providing evidence to support Einstein's general relativity.

To draw the image of the gravitational lens, really just a skybox texture is required. Since Galaxia already renders the skybox for star systems, this is reused here. However, although a star system is quite small compared to the distance to nearby stars, the closest stars might appear to be in the wrong locations when drawn in the skybox since it is static over the entire star system. To counter this effect, usually Galaxia does not render nearby stars into the skybox, but instead renders them as normal objects afterwards. This would mean that some stars might appear to be missing in the gravitational lens, so for the black hole systems the engine must render all the stars into the skybox. This unfortunately does reintroduce the star position "popping" effect as the skybox transitions to the full rendering of the stars, when entering or leaving the system. But these are unfortunately the sorts of prices we pay for real-time rendering on consumer hardware. The solution might be to render a separate skybox for the gravitational lens, but this would require a lot more graphics memory. For the moment at least, I'll put up with the popping effect. It's not really that noticeable anyway if you're not paying close attention.

With the skybox texture in place, some geometry needs to be rendered to generate some pixels. The same as some other volumetric effects I've been working on, I start with an inverted cube, which is 100 times the **Schwarzchild radius (https://en.wikipedia.org/wiki/Schwarzschild_radius)** (in the

case of a black hole, the Schwarzchild radius is the proper radius of the event horizon, or “shadow”). It is also worth noting that the Schwarzchild radius equation is closely related to the gravitational lens equation (image from **wikipedia (https://en.wikipedia.org/wiki/Schwarzschild_radius)**):

$$r_s = \frac{2GM}{c^2}$$

So, if the black hole’s Schwarzchild radius is calculated as part of the procedural generation system, the deflection of each pixel’s ray can be easily calculated as 2*rs/b:

```
float rs = SchwarzchildRadius; //constant for the shader. depends on the mass
float3 p = BlackHolePosition; //relative to the camera. constant for the shader
float3 rd = normalize(input.Direction); //ray direction for the pixel.
float b = PointRayDist(p, rd); //impact parameter.
float a = 2 * rs / b; //deflection angle - thanks Albert!
```

Where b is the impact parameter, calculated using the point-ray distance function:

```
float PointRayDist(float3 p, float3 rd)
{
   return length(cross(rd, p));
}
```

Where p is the position of the black hole relative to the camera, and rd is the ray direction. Note that this function assumes that the ray passes through the origin.

I also subtract a small constant value from the calculated deflection angle so that the edges of the lens can blend smoothly with the existing background skybox. The deflection would actually be visible out to many hundreds or even thousands of times the Schwarzchild radius, but for real-time rendering purposes this is not really practical. I had chosen the rendered inverted cube size to be 100 times the Schwarzchild radius to try and keep the result as close to reality as practical. The small offset constant was chosen to work well with that size cube (in case you're wondering, the value I use is 0.02).

Now that the deflection angle is known, the deflected ray direction can be calculated. I have done this by using a quaternion:

```
float3 bhdir = normalize(p); //the direction to the center of the black hole
float3 axis = normalize(cross(rd, bhdir)); //the axis of deflection
float4 qdef = qaxang(axis, a); //quaternion representing the deflection
float3 ddir = mulvq(rd, qdef); //apply the rotation to the original ray dir
```

(See my **galaxy rendering post (https://dexyfex.com/2016/09/09/galaxy-rendering-revisited/)** for the definitions of qaxang and mulvq functions)

The deflected direction vector is then used to sample the skybox as normal.

This still won't produce an image of a black hole though, just a gravitational lens approximation. To get the shadow to appear, a simple solution is to choose a maximum deflection value, and output black for any angle higher than that. On comparison with the actual Schwarzchild radius, I found that a maximum deflection angle of pi/2 (90 degrees) comes close to being correct, while preserving some of the stronger distortions. To reduce aliasing along the shadow's edge, I blend the black shadow depending on the deflection angle as it approaches the limit. While not strictly realistic, it's a sacrifice made in the name of image quality.

Even though this is a very cheap and quite inaccurate approximation for a black hole's gravitational lens and shadow, the results are fairly convincing – showing the distinct **Einstein ring (https://en.wikipedia.org/wiki/Einstein_ring)** effect, and the upward curvature of the shadow as the camera passes the event horizon, resulting in the "engulfment" effect:

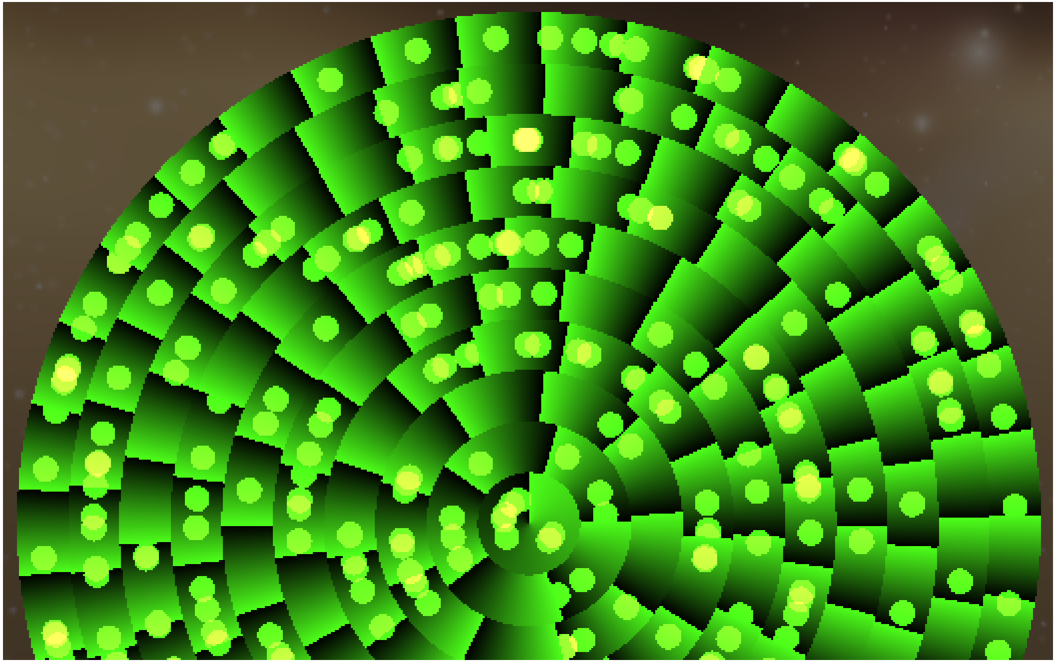
Flying into a black hole (and out again) - Galaxia…

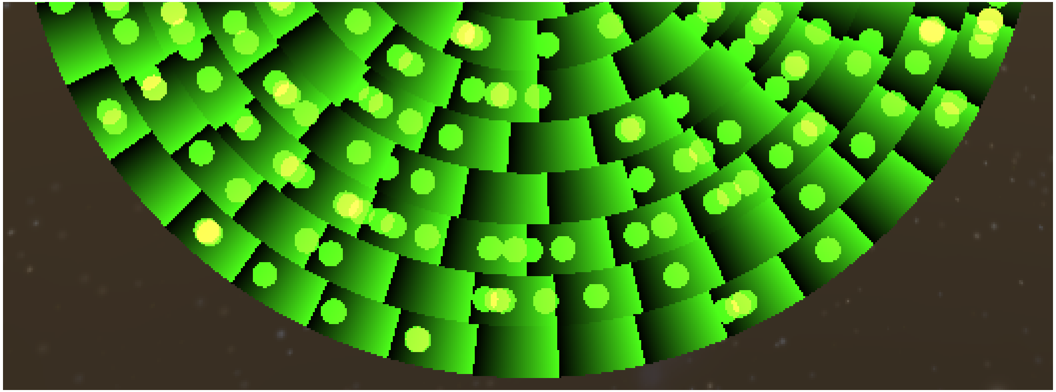
Some special considerations need to be taken into account when looking out from within the event horizon. Since the full field of view is compressed into a tiny point, the intensity of light is magnified, so I multiply the colour output by an arbitrary factor based on the deflection angle to provide this aesthetic effect. This works well since I am using floating-point render targets and HDR post processing.

Most good black holes apparently have accretion discs around them, so this can't be overlooked. Neutron stars will often have them too, as well as the visible gravitational lens due to their incredible density, so I should be able to re-use the technique again later.

Since I'm already doing lensing calculations in this shader, I thought it would be simplest just to add the sampling of an accretion disc texture based off the same deflection calculations. The first step with this approach is to generate a texture. I wanted the accretion disc to be able to spin, and to have the inner areas appearing to be spinning faster than the outer ones. Unfortunately the standard sort of Perlin noise can't really do this – if we sample Perlin noise with increasing rotation in some domain warping (think of "twisting" the noise), the noise function will "wind up" over time, to a point where the noise would end up just looking like a set of concentric circles.

Taking a similar approach to the dust clouds in the version 2 of my galaxy rendering, my “spinning noise” works by randomly positioning a series of circles within a known range and summing each one’s contribution depending on its distance from the sample point. From this point of view, it is a variant of cell noise (aka **Worley noise (https://en.wikipedia.org/wiki/Worley_noise)**). The major difference is however that the grouping of the circles is performed in **polar coordinates (https://en.wikipedia.org/wiki/Polar_coordinate_system)**. This is perhaps best explained with an image:





Spinning noise rings and segments, containing example particles

Each concentric ring is divided into segments depending on the radius, to get approximately equal-sized segments. In each segment, n data points are randomly positioned. The concentric rings are then made to rotate at varying speeds, with the inner rings spinning faster. The main advantage of this method is that for each pixel in the final image, only 4 segments need to be sampled, regardless of how many concentric rings or segments there are. For any given pixel, the polar coordinates and therefore the corresponding ring and segment are easily calculated. If we assume the circles can overlap the ring and segment borders only in one direction, only the current cell and the 3 others providing a possible overlap are the cells that need to be sampled for each pixel.

Here's the animated final effect – if you look hard enough you will be able to just notice the concentric rings pattern. But don't stare at it too long or too closely because when you look away from it everything will look like it's spinning!! Don't say I didn't warn you!